Part 8

Intermediate Code

Compiler Output

- A compiler ultimately has to make output
  - Assembly code
  - C code
  - Virtual machine instructions
- Yes, we've worked on that.

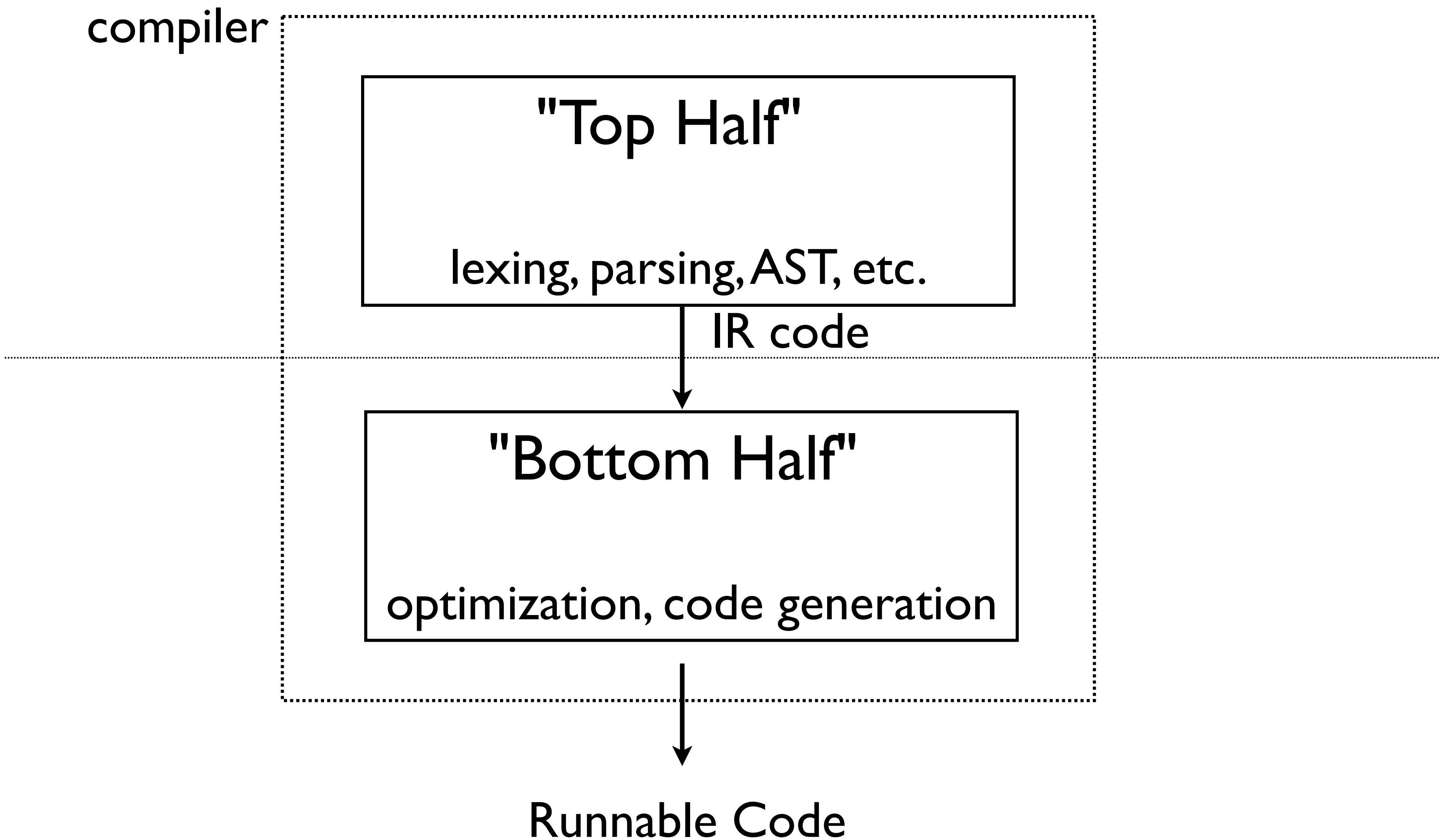
Backing up....

- Writing a compiler is hard
- Do you make it target just one thing?
- For example, a single model of a CPU?
- Usually not
- There is an abstraction of "hardware"

Intermediate Code

- Compilers often prefer to generate an abstract intermediate code instead of directly emitting low-level HW instructions or C code.
- Intermediate code is sort of a generic "machine code"
- This abstract code is easier to analyze, optimize, transform, etc.

Compiler Design



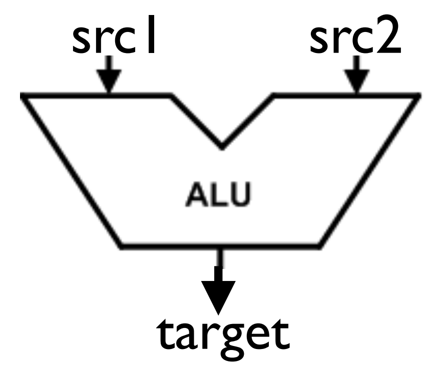
Designing an IR

- Intermediate Representation is usually modeled after actual computer hardware
- Registers, memory, low-level operations
- However, a lot of constraints are removed (for example, infinite registers)

Three-Address Code

- A common IR where most instructions are tuples `(op,src1,src2,target)`

```
('ADD', a, b, c)        # c = a + b
('SUB', x, y, z)        # z = x - y
('LOAD', a, b)          # b = a
```



- Closely mimics design of an ALU
- The initial warmup exercise was 3AC

Three-Address Code

- Example of three-address code IR

```
c = 2*a + b - 10

    r1 = 2                 ('CONST',2,'r1')
    r2 = a                 ('LOAD','a','r2')
    r3 = r1 * r2           ('MUL','r1','r2','r3')
    r4 = b                 ('LOAD','b','r4')
    r5 = r3 + r4           ('ADD','r3','r4','r5')
    r6 = 10                ('CONST',10,'r6')
    r7 = r5 - r6           ('SUB','r5','r6','r7')
    c = r7                 ('STORE','r7','c')
```

- Calculations are broken down into steps that carry out one operation at a time

SSA Code

- Single Static Assignment
- A restriction of 3-address code
    - Infinite registers
    - Registers are immutable
    - Can never reassign a previously used register
- This is basis of systems such as LLVM

Stack Machine

- Another common IR abstraction: get rid of registers altogether and use a stack
- General idea
  - Operands get pushed onto stack
  - Operators consume stack items
  - Result gets pushed back onto stack

Example : Stack Machine

- Example: Compute: 2 + 3 * 4

Instructions	Stack
`PUSH 2`	`[2]`
`PUSH 3`	`[2, 3]`
`PUSH 4`	`[2, 3, 4]`
`MUL`	`[2, 12]`
`ADD`	`[14]`

- Common stack machines
  - JVM (Java)
  - Python
  - WebAssembly

Code Generation

- Code generation for either option is often not much more than a traversal of the program structure (data model)
- Walk the nodes and emit instructions
- Will be very similar to type-checking

Example (Stack Machine)

```
def emit_Integer(node, code):
    code.append(('PUSH', node.value))

def emit_BinOp(node, code):
    emit(node.left)
    emit(node.right)
    if node.op == '+':
        code.append(('ADD',))
    elif node.op == '-':
        code.append(('SUB',))
    elif node.op == '*':
        code.append(('MUL',))
    ...
```

Digression

- Stack machines can be converted into 3AC

Stack IR	3AC	Stack
`PUSH 2`	`('CONST', 2, 'r1')`	`[r1]`
`PUSH 3`	`('CONST', 3, 'r2')`	`[r1, r2]`
`PUSH 4`	`('CONST', 4, 'r3')`	`[r1, r2, r3]`
`MUL`	`('MUL','r2','r3','r4')`	`[r1, r4]`
`ADD`	`('ADD','r1','r4','r5')`	`[r5]`

- Use a stack to track the register names
- Emit 3AC codes with names from stack

Structured Control Flow

- IR might also use structured control flow

```
if a < b {
   statements1
} else {
   statements2
}
```

current block

```
...
LOAD a
LOAD b
LT
IF
statements1
ELSE
statements2
ENDIF
...
```

Create instructions to express the block-structure of the conditional

Structured Control Flow

- Example of a loop

```
while a < b {
   statements1
}
statements2
```

```
LOOP
LOAD a
LOAD b
GE
CBREAK
statements1
ENDLOOP
statements2
```

- Looks a bit funny, but it's same idea as this

```
while True {
   if not (a < b) {
      break
   }
   statements1
}
statements2
```

The World of Registers

- IR is where a lot of optimization takes place
  - Identifying repeated patterns
  - Reusing values
  - Optimal register allocation
- Frankly, this is a LOT of magic (topic of a more advanced compiler course)

Optimization

- Example: peephole optimization

```
r1 = 2
r2 = a
r3 = r1 * r2
r4 = b
r5 = r3 + r4
r6 = 10
r7 = r5 - r6
c = r7
r8 = 10
r9 = c
r10 = r8 * r9
d = r10
```

- A search for unnecessary instructions

Optimization

- Example: peephole optimization

```
r1 = 2
r2 = a
r3 = r1 * r2
r4 = b
r5 = r3 + r4
r6 = 10
r7 = r5 - r6
c = r7
r8 = 10
r9 = c
r10 = r8 * r9
d = r10
```

repetition of values

- A search for unnecessary instructions

Optimization

- Example: peephole optimization

```
r1 = 2
r2 = a
r3 = r1 * r2
r4 = b
r5 = r3 + r4
r6 = 10
r7 = r5 - r6
c = r7
r8 = 10
r9 = c
r10 = r8 * r9
d = r10
```

Can delete redundant register

```
r1 = 2
r2 = a
r3 = r1 * r2
r4 = b
r5 = r3 + r4
r6 = 10
r7 = r5 - r6
c = r7
------
r9 = c
r10 = r6 * r9
d = r10
```

- A search for unnecessary instructions

Optimization

- Example: peephole optimization

```
r1 = 2
r2 = a
r3 = r1 * r2
r4 = b
r5 = r3 + r4
r6 = 10
r7 = r5 - r6
c = r7
------
r9 = c
r10 = r6 * r9
d = r10
```

Wasteful store/load

```
r1 = 2
r2 = a
r3 = r1 * r2
r4 = b
r5 = r3 + r4
r6 = 10
r7 = r5 - r6
c = r7
------
------
r10 = r6 * r7
d = r10
```

- A search for unnecessary instructions

Optimization

- Example: Subexpression elimination

```
(x+y)/2 + (x+y)/4
```

```
r1 = x
r2 = y
r3 = r1 + r2
r4 = 2
r5 = r3 / r4
r6 = x
r7 = y
r8 = r1 + r2
r9 = 4
r10 = r8 / r9
```

→

```
r1 = x
r2 = y
r3 = r1 + r2
r4 = 2
r5 = r3 / r4



r9 = 4
r10 = r3 / r9
```

- Only safe if x/y guaranteed not to change

Packaging of IR code

- Intermediate code is <u>more</u> than instructions
  - Modules
  - Functions
  - Variables
- There is a lot of metadata associated with the instructions (names, types, scopes, etc.)

Functions

- Instructions are attached to a function
- There is metadata (name, types, etc.)

```
func bar(a int, b int) int {
    var z int;
    z = a * b;
    return z;
}
```

- You will make function "objects"

<u>function</u>:

```
name: "bar"
parameters: [int, int]
returns: int
locals: [int]
code:  [
     ('LOAD', 'a', 'r1')
     ('LOAD', 'b', 'r2')
     ('MUL', 'r1', 'r2', 'r3')
     ('STORE', 'r3', 'z')
     ...
     ]
```

Modules

- A "module" is the product of compilation
- A container for the compiled declarations

```
function: foo
function: bar
global: x
...
```

- It is helpful to think of Python here. Python code is organized into modules. A module represents a file of source code. Modules contain definitions of functions, variables, classes. Compilers do something similar.

Project

Tutorial: LLVM Tutorial / Wasm Tutorial

Find the file `wabbit/llvm.py`

Find the file `wabbit/wasm.py`

Follow the instructions inside